UNIVERSITÀ DEGLI STUDI DI MESSINA

DIPARTIMENTO DI SCIENZE MATEMATICHE E INFORMATICHE, SCIENZE FISICHE E SCIENZE DELLA TERRA

Corso di Laurea Triennale in Informatica

Confronto
MongoDB vs Neo4j

Docente: Antonio Celesti

Studente: Thilina Perera P.A (514553)

---

[0]Anno Accademico: 2022/23

progetto NoSql

Thilina Perera P.A

September 2023

Contents

1 Obiettivi

Il progetto consiste nello studio e nel confronto tra due DBMS NoSQL: MongoDB e Neo4j. Di entrambi i database verrà esposto il funzionamento e il dataset impiegato.

I database verranno confrontati sui tempi di esecuzione di quattro query di complessità crescente, tratte da un caso di studio, effettuate su un dataset di dimensione crescente (25%, 50%, 75% e 100% del dataset originale).

I risultati ovvero i tempi sono misurati in millisecondi e sono stati poi scritti su file csv, importati su una cartella di lavoro Excel e opportunamente organizzati in tabelle e grafici.

Il progetto è stato implementato usando Docker per creare le istanze dei database e Python per automatizzare la creazione del dataset, l'inserimento dei dati nei database, l'esecuzione delle query e la raccolta dei risultati.

Tutti i test sono stati effettuati su una macchina con Microsoft Windows 11 home, un processore AMD Ryzen 7 e 16GB di RAM 8 core

Il codice sorgente e visionabile sul repository GitHub[1]

[1] https://github.com/Thi-Perera/neo4j-mongodb-performance-analysis

2 Introduzione

I database NoSQL sono nati per ovviare a mancanze che i database relazionali (RDBMS, Relational DBMS) avevano, ovvero capacità di immagazzinare, organizzare e interagire con grandi quantità dati senza che vi sia uno schema rigido, e in tempi accettabili per le varie necessità dell'utenza. Ciò portò allo sviluppo di nuovi paradigmi, come il keyvalue, a grafo o column-based (3 tipologie di Db nosql diversi), ognuno con caratteristiche e implementazioni differenti, prestazioni migliori o peggiori dipendentemente da casi e circostanze.Alcuni Esempi per ognuno delle categorie sopracitate sono: MongoDB, etichettato più precisamente come document-oriented, Neo4j db a grafo e Cassandra column-based. Tutti i DBMS rispettano il teorema di Brewer, o Teorema CAP, per cui un sistema informatico distribuito non può soddisfare tutte e tre le seguenti garanzie: • Consistenza (Consistency); • Disponibilitá (Availability); • Partizionamento (Partitioning). Difatti, esempi come i database key-value rispettano la sigla CP, consistenza e partizionamento, mentre i database a grafo rispettano la sigla CA, consistenza e disponibilità (le stesse caratteristiche ACID dei database relazionali). In questo studio vedremo queste esatte caratteristiche in gioco, senza aver a che fare in alcun modo con il partizionamento e agevolando la disponibilità perchè tutte le prestazioni analizzate sono misurate su una distribuzione su singolo nodo.

3 Database

I due DBMS scelti per questa comparazione sono MongoDB e Neo4j, per semplicità di implementazione e per i due paradigmi diversi, deployment tramite containers e interfacciamento con l'utente via riga di comando.

entrambi containerizzati con la funzione docker compose e il seguente file yaml:

```yaml
version: '3'
services:

  mongodb:
    container_name: "mymongodb_container"
    image: "mongo"
    ports:
      - 27017:27017

  neo4j:
    container_name: "myneo4j_container"
    image: "neo4j"
    ports:
      - 7474:7474
      - 7687:7687
    environment:
      - NEO4J_AUTH=neo4j/neo4jnosqlproject
```

3.1 Primo Database: Mongodb

MongoDB è un DBMS di tipo NoSQL document-oriented e schema-free: la prima etichetta indica che la più piccola porzione di informazione ricavabile dai dati è chiamato documento; la seconda evidenzia come questi ultimi non necessitino di alcun tipo di schema rigido o semi-rigido. Un documento di MongoDB è una porzione di dati archiviati come una coppia key-value, così da permettere un accesso al dato da estrapolare diretto, con complessità computazionale O(1), e senza alcun tipo di vincolo di coerenza di forma o schema tra documenti della stessa collezione (raccolta di documenti, analogo alle tabelle dei RDBMS). Inoltre, il formato dei documenti, detto BSON (Binary Serialized dOcument Notation), è derivato dal formato testuale JSON (JavaScript Object Notation), e ne rappresenta il corrispettivo serializzato come dato binario. Per questa sua

tipologia, un documento in formato BSON gode di una più veloce elaborazione. Una caratteristica rilevante di MongoDB, è la possibilità di dividere fisicamente i dati su più nodi distinti (sharding), sui quali eseguono dei replica-set (nodi repliche di mongodb), ciò permette una elaborazione parallela(parallelizzazione) delle operazioni e fault-tolerance alla caduta di uno dei nodi. Tutto ciò è possibile per via della gestione ad un più alto livello dei dati in i documenti e collezioni. Un database document-oriented come MongoDB eccelle nella ricerca dei dati per via delle coppie key-value e dei calcoli paralleli sui più nodi del database distribuito(parallelizzazione non provata nello studio, è stato usato un solo nodo), peccando però dell'impossibilità di JOIN ottimizzati tra più documenti o collezioni, Adesso è possibile grazie alla funzionalità aggregate aggiunta solo in versioni più recenti del DBMS.

è stato istanziato un container partendo dall'immagine mongo2 ufficiale di Docker; si è anche pubblicata la porta 27017, di default quella attraverso la quale il DBMS resta in attesa di connessione, cosicché sia possibile permettere un'interazione con il database, nel nostro caso da parte degli script Python precedentemente citati.

3.2 Secondo database: Neo4j

Neo4j è un DBMS NoSQL a grafo, ovvero i dati sono organizzati secondo l'omonima struttura dati. Difatti, la minima porzione di informazione ricavabile è un nodo del grafo, i cui dati sono le proprietà del nodo; analogamente le relazioni sono effettivamente gli archi del grafo, sempre orientati, e anch'esse con proprietà, ove necessario. Questa struttura dati permette di sfruttare tutti i vantaggi di un grafo, come gli attraversamenti e le ricerche, effettuabili sull'intero grafo o su sottografi, combinati agli accessi alle proprietà di nodi e relazioni, implementate come coppie key-value, con accesso diretto. Inoltre, anche i database a grafi sono schema-free, in quanto ogni nodo può possedere proprietà differenti da altri nodi della stessa categoria (label secondo il linguaggio di Neo4j, esempio un nodo etichettato come person e employee). I database a grafi sono i più versatili e godono di essere, generalmente, i più prestazionali quando vi è l'esigenza di tener conto di molte relazioni tra varie entità, inciampando però nell'impossibilità di partizionamento del grafo su più nodi(computers).

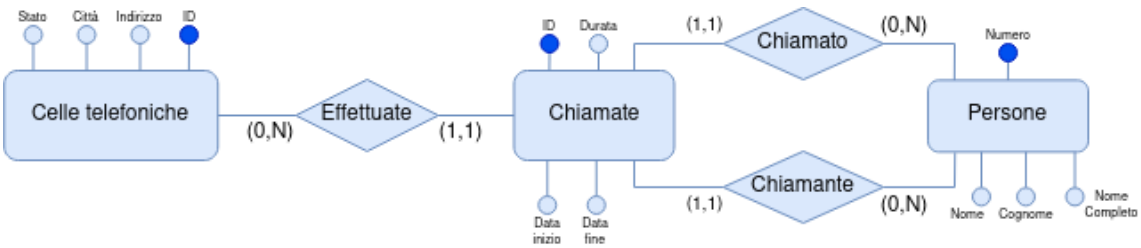
Il container è istanziato a partire dall'immagine ufficiale neo4j 3 e sono state pubblicate due porte: la 7474, attraverso la quale viene messa a disposizione un'interfaccia web per l'interazione con il database, e la 7687, attraverso la quale il DBMS resta in ascolto in attesa di connessione esterna per l'interazione con il database.m Inoltre, in questo caso è stato necessario l'utilizzo di un volume, o più precisamente il binding del filesystem dell'host con quello del container, per permettere la popolazione del database tramite file esterni; come sopra, ciò non è necessario ai fini dello studio.

4 Progettazione

4.1 Caso studio

Il Tema di sottofondo affrontato in questo studio è la ricerca di criminali tramite delle operazioni di incrociamento di chiamate, date e luoghi di interesse, al fine di trovare contatti che questi possono avere o aver avuto, o quantomeno di restringere il campo di ricerca. Un possibile scenario potrebbe riguardare l'indagine su un gruppo di criminali coinvolti in un furto. Immaginiamo di voler condurre un'indagine basata sulla ricerca delle chiamate telefoniche effettuate nelle celle telefoniche situate nelle zone circostanti al luogo del crimine e in determinate date. L'obiettivo potrebbe essere quello di individuare le persone che hanno effettuato o ricevuto chiamate in prossimità della scena del crimine, con la speranza di identificare una cerchia ristretta di individui che potrebbero essere coinvolti nell'attività criminale.

Supponendo di avere i permessi legali dei gestori telefonici di memorizzare alcuni dati inerenti le chiamate, si possa costruire un database con uno schema concettuale corrispondentemente a quello che segue:



Le entità presenti nel database possono essere suddivise in tre categorie:

• Persone, caratterizzate da un numero di telefono univoco, un nome, un cognome, e nome completo;

• Chiamate, caratterizzate da un ID univoco, data inizio chiamata e data fine chiamata, dove per data si intende la codifica UNIX Epoch dell'istante corrente, una durata, le due persone comunicanti e la cella telefonica nella quale è effettuata;

• Celle telefoniche, caratterizzate da un ID univoco, uno Stato, banalmente l'Italia nel nostro esempio, una città e una via dove queste sono localizzate.

Le relazioni che si osservano sono le seguenti:

• Chiamante/Chiamato, che legano le persone alle chiamate da loro effettuate/ricevute;

• Effettuate, che lega le chiamate alla cella telefonica nella quale sono effettuate.

4.2 MongoDB

Su MongoDB, le entità, diventate collezioni, sono le seguenti tre, con i relativi campi:

• person:

– Number, numero di telefono;

– FullName, nome completo;

– FirstName, nome;

– LastName, cognome;

• call:

– CallingNbr, numero del chiamante;

– CalledNbr, numero del chiamato;

– StartDate, momento di inizio chiamata;

– EndDate, momento di fine chiamata;

– Duration, durata;

– CellSite, ID della cella;

• cell:

– CellSite, ID della cella;

– City, Città;

– State, Stato (Italia);

– Address, indirizzo stradale;

Come ID della collezione call ci si è avvalsi dell'Document ID che MongoDB assegna automaticamente al momento dell'inserimento del documento. Le relazioni, invece, non sono esplicitabili con questo DBMS, ma, qualora ve ne sia il bisogno, è necessario che sia il progettista a ideare il database(con referencing o embending ad esempio) e le interfacce con esso in maniera tale che vi sia modo di effettuare delle query aggregate per concretizzare le relazioni concettuali.

4.3 Neo4j

In Neo4j, le entità, diventate labels, sono le seguenti tre, con relative proprietà dei nodi:

• person:

– Number, numero di telefono;

– FullName, nome completo;

– FirstName, nome;

– LastName, cognome;

• call:

– CallingNbr, numero del chiamante;

– CalledNbr, numero del chiamato;

– StartDate, momento di inizio chiamata;

– EndDate, momento di fine chiamata;

– Duration, durata;

– CellSite, ID della cella;

• cell:

– CellSite, ID della cella;

– City, Città;

– State, Stato (Italia);

– Address, indirizzo stradale.

Analogamente a MongoDB, gli ID dei nodi call sono rappresentati dagli id che Neo4j assegna alla creazione del nodo. Le relazioni, a differenza del precedente DBMS, sono esplicitabili come archi del grafo, e sono le seguenti tre:

• is_calling, diretto da un nodo person verso un nodo call;

• is_called, diretto da un nodo call verso un nodo person;

• is_done, diretto da un nodo call verso un nodo cell.

5 Implementazione

5.1 Generazione dati

Per l'ottenimento dei dati che andranno a popolare i database, è stato scritto uno script in Python che generasse pseudo-casualmente tramite le API del modulo Faker. Lo script presenta tre funzioni separate per la generazione dei dati delle tre entità previste, a ciascuno si effettua una scrittura su file csv (Comma Separated Values), creato, o sovrascritto se esistente, runtime. Le tre funzioni vengono invocate su due threads, in maniera tale da eseguire concorrentemente la generazione delle persone e delle celle telefoniche, per poi attenderne la fine e richiamare la generazione delle chiamate. Tale implementazione è stata scelta per avere un minore tempo di esecuzione (non realmente necessario), e per la necessità di eseguire separatamente l'ultima delle tre funzioni per via della dipendenza dell'entità chiamate dalle altre due. La quantità di dati generata, come da traccia, è pari al 25un massimo deciso, esprimibile al codice come parametro al momento di esecuzione da riga di comando; di seguito la tabella dei dataset:

	25%	50%	75%	100%
Person	5.000	10.000	150.000	20.000
cells	4.000	8.000	12.000	16.000
calls	250.000	500.000	750.000	1.000.000
totale	259.000	518.000	912.000	1.036.000

Figure 1: Descrizione dell'immagine

Di seguito una vista della sezione dedicata al multithread, tramite modulo threding:

```
thread = Thread(target=gen_cells, args=(num_cells,))
thread.start()
gen_people(num_people)
thread.join()
gen_calls(num_calls, num_people, start_date, end_date,range_call)
```

5.2 Inserimento dati

Anche per la popolazione dei database è stato utilizzato uno script Python, con implementazione multithread. Le funzioni utilizzate sono fornite dai driver open-source pymongo e neo4j per Python. La connessione ai due database è effettuata tramite valori hardcoded di indirizzo IP e porta (127.0.0.1, 27017 per MongoDB e 7687 per Neo4j) ad hoc per la macchina di sviluppo e testing, considerando i due DBMS distribuiti tramite Docker images, come descritto. Di seguito una vista della sezione dedicata al multithread, tramite modulo threding:

```
if mongo:
if not neo:
insert_mongo(debug=debug)
else:
thread_mongo = Thread(target=insert_mongo,
kwargs={'debug': debug})
thread_mongo.start()
if neo:
insert_neo(debug=debug)
if 'thread_mongo' in locals():
thread_mongo.join()
```

5.3 testing dei Database

Le queries che sono state effettuate sui due database sono 4 e seguono il seguente schema, per semplicità descritto in sintassi SQL, caratterizzato da una difficoltà computazionale crescente:

1) 2 WHERE;

2) 2 WHERE 1 JOIN;

3) 2 WHERE 2 JOIN;

4) 3 WHERE 2 JOIN.

Ciò è scelto per mostrare come i due DBMS si comportino al crescere della complessità delle interrogazioni e delle dimensioni dei dataset. L'obiettivo principale di questo studio è valutare e confrontare le prestazioni di due software database, MongoDB e Neo4j, mentre gestiscono crescenti volumi di dati e interrogazioni computazionalmente impegnative. L'analisi mira a identificare i punti di forza e di debolezza di entrambi i sistemi in queste condizioni di stress.

Per condurre il confronto, le query non vengono eseguite utilizzando le interfacce CLI integrate dei due DBMS (mongosh e cypher-shell per MongoDB e Neo4j rispettivamente). Invece, vengono utilizzati script Python che implementano gli stessi moduli descritti precedentemente (5.2). Gli script includono anche la registrazione automatica dei tempi di esecuzione, attivabili tramite flag al momento dell'esecuzione del codice da terminale.

I risultati delle misurazioni vengono quindi registrati in file CSV secondo il seguente formato:

Tempo alla prima esecuzione Tempi delle 30 esecuzioni successive Media dei tempi delle 30 esecuzioni Deviazione standard (misura di dispersione o variabilità in un set di dati) del campione misurato Dopo la prima esecuzione in un ambiente appena configurato, sono state eseguite ulteriori 30 misurazioni per ciascuna query al fine di valutare le prestazioni dei meccanismi di caching implementati dai due DBMS.

I dati raccolti sono stati successivamente importati manualmente in Microsoft Excel per elaborare due tipi di istogrammi che consentono una visualizzazione chiara dei risultati. In particolare:

1. Un istogramma che confronta i tempi di esecuzione della prima esecuzione di ciascuna query, evidenziando le differenze tra MongoDB e Neo4j. 2. Un secondo istogramma che mette a confronto le medie dei tempi di esecuzione delle 30 esecuzioni successive, con annesso un intervallo di confidenza al 95

Tutti i grafici presentano i dati relativi alla query in esame per tutti e quattro i dataset considerati.

Le funzioni sviluppate per eseguire le query nei due database sono state implementate come segue:

```
# MongoDB
def exec_query(query, client=connect_mongo(), t=False):
    start = timestamp()
    client.test.calls.aggregate(query)
    return timestamp(start) if t else 0
# Neo4j
def exec_query(query, client=connect_neo(), t=False):
    start = timestamp()
    with client.session() as db:
    db.run(query)ty
    return timestamp(start) if t else 0
```

La prima funzione, che si occupa dell'esecuzione delle query su MongoDB, opera tramite il metodo aggregate() fornito dai moduli sopracitati, il quale corrisponde all'omonima funzione built-in del DBMS; questa necessita di una lista di dizionari (strutture dati native di Python), i quali corrispondono alle varie operazioni che devono essere effettuate sui dati, secondo un meccanismo di pipeline. Si è utilizzata questa funzione in tutte le query, nonostante non fosse sempre necessaria

e fosse sostituibile con la più consueta funzione find(), per mantenere una coerenza fra tutte; inoltre, questa è l'unica modalità di interrogazione dove è presente un'operazione "JOIN" (secondo un'analogia con il linguaggio SQL). La seconda funzione, che si occupa dell'esecuzione delle query su Neo4j, utilizza il metodo run() fornito dai moduli citati, il quale opera semplicemente tramite una stringa contenente l'interrogazione. Questo è valido per una qualsiasi esecuzione di query su Neo4j. Entrambe le funzioni ritornano, se opportunamente indicato tramite parametro, il tempo di esecuzione della query, altrimenti 0 per indicare una corretta esecuzione.

5.3.1 Query 1

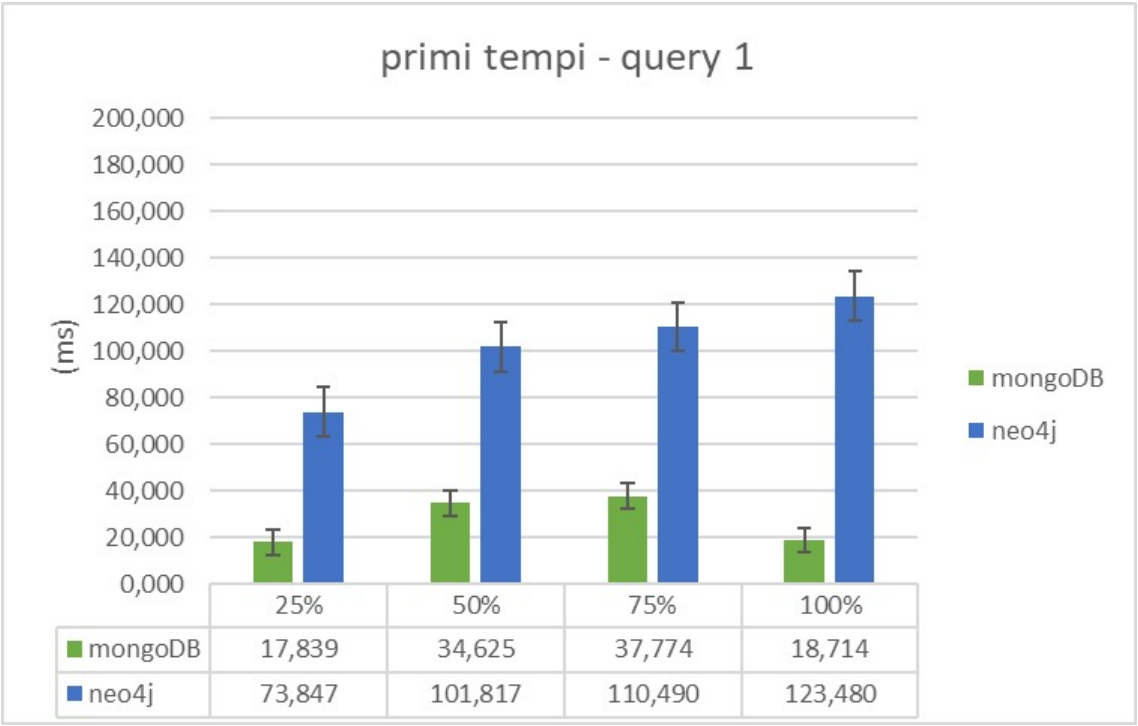
La prima query effettuata presenta una selezione condizionata con 2 WHERE su singola entità; di seguito le corrispondenti versioni:

```
# SELECT *
# FROM calls
# WHERE StartDate >= 1580083200 AND EndDate < 1672473600
# MongoDB
start_search = int(mktime(datetime(2020, 1, 27).timetuple()))
end_search = int(mktime(datetime(2020, 1, 29).timetuple()))
query = [
{"$match": {"StartDate": {"$gte": start_search,
"$lt": end_search}}}
]
# Neo4j
query = "MATCH (c:call) \
WHERE c.StartDate >= 1580083200 \
AND c.StartDate < 1672473600 \
RETURN c"
```

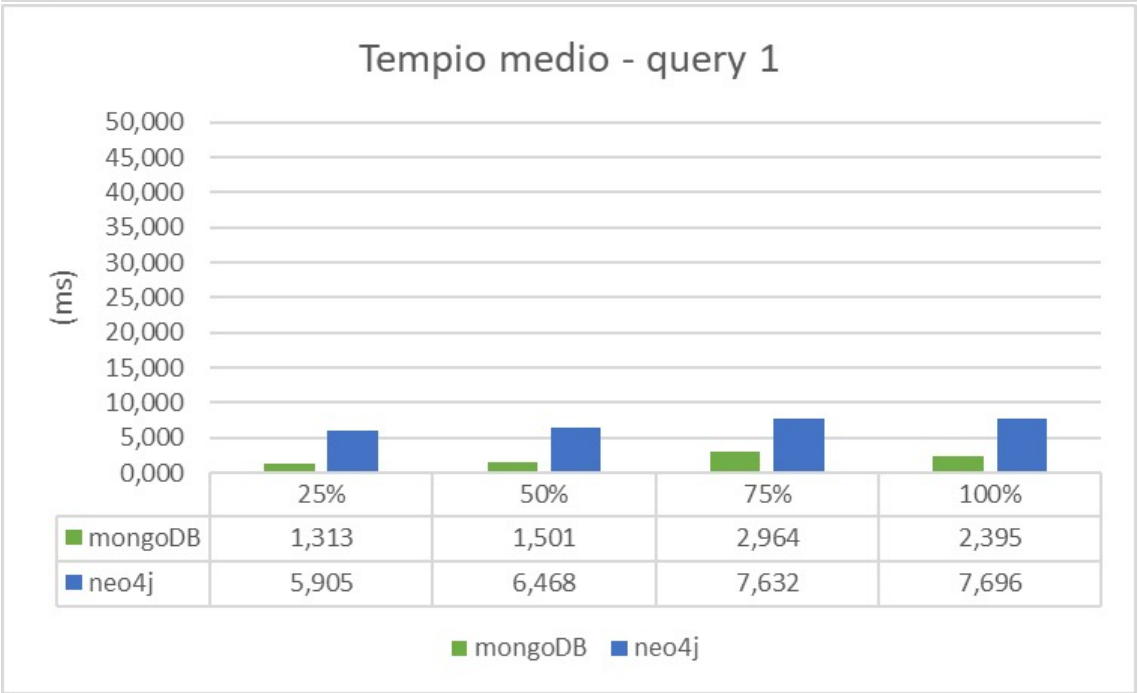
Il numero "1580083200" rappresenta un timestamp Unix. Un timestamp Unix è una rappresentazione numerica del tempo che conta i secondi trascorsi dal 1 gennaio 1970 alle 00:00:00 UTC (Coordinated Universal Time), noto anche come "Epoch Unix". Mentre "1672473600" è 1 gennaio 2023.

I tempi di esecuzione misurati (in millisecondi) sono i seguenti:

	primi tempi			media	
	mongoDB	neo4j		mongoDB	neo4j
25%	17,839	73,847	25%	1,313	5,905
50%	34,625	101,817	50%	1,501	6,468
75%	37,774	110,490	75%	2,964	7,632
100%	18,714	123,480	100%	2,395	7,696



	25%	50%	75%	100%
mongoDB	17,839	34,625	37,774	18,714
neo4j	73,847	101,817	110,490	123,480



	25%	50%	75%	100%
mongoDB	1,313	1,501	2,964	2,395
neo4j	5,905	6,468	7,632	7,696

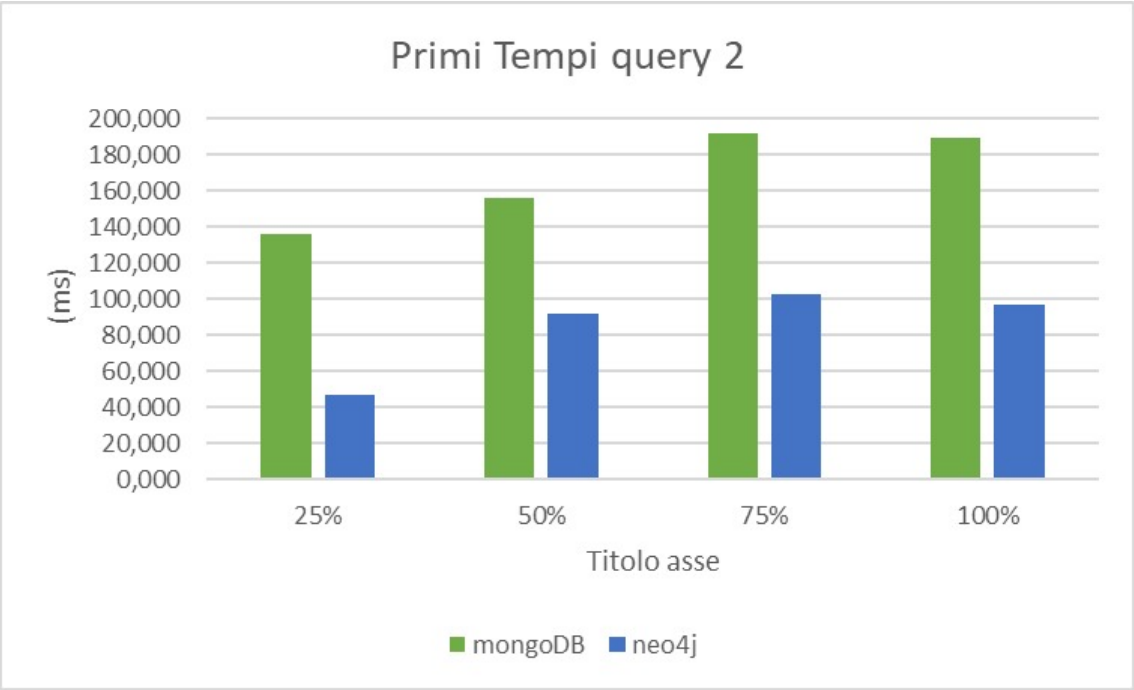
5.3.2 Query 2

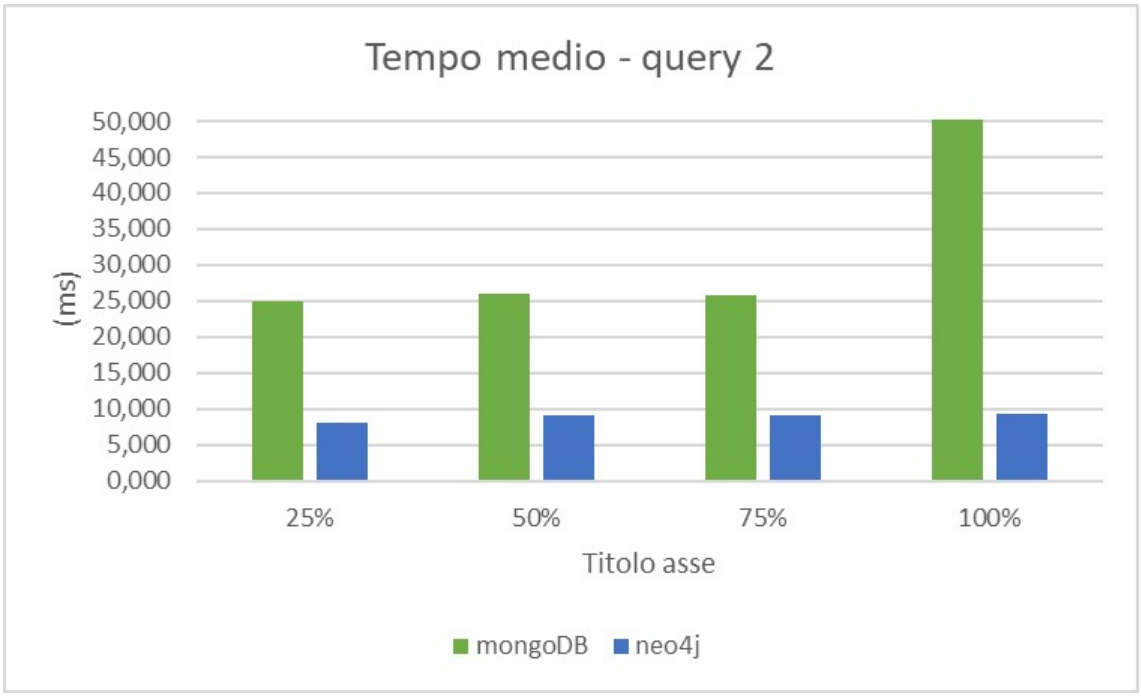
La seconda query effettuata presenta una selezione condizionata con 2 WHERE su due entità, unite tramite 1 JOIN; di seguito le corrispondenti versioni:

```
# SELECT *
# FROM calls
# JOIN people
# ON calls.CallingNbr = people.Number
# WHERE StartDate >= 1580083200 AND EndDate < 1672473600
# MongoDB
query =[
{"$match": {"StartDate": {"$gte": start_search,
"$lt": end_search}}},
{"$lookup": {"from": "people",
"localField": "CallingNbr",
"foreignField": "Number",
"as": "Calling"}}
]
# Neo4j
query = "MATCH (p1:person)-[r1:is_calling]->(c:call) \
WHERE c.StartDate >= 1580083200 \
AND c"
```

I tempi di esecuzione misurati (in millisecondi) sono i seguenti:

	primi tempi			media	
	mongoDB	neo4j		mongoDB	neo4j
25%	136,422	46,847	25%	24,959	8,152
50%	156,174	91,817	50%	26,116	9,130
75%	192,353	102,490	75%	25,775	9,068
100%	189,636	96,480	100%	68,912	9,279




Tempo medio - query 2
50,000
45,000
40,000
35,000
30,000
25,000
20,000
15,000
10,000
5,000
0,000
(ms)
25%
50%
75%
100%
Titolo asse
mongoDB
neo4j

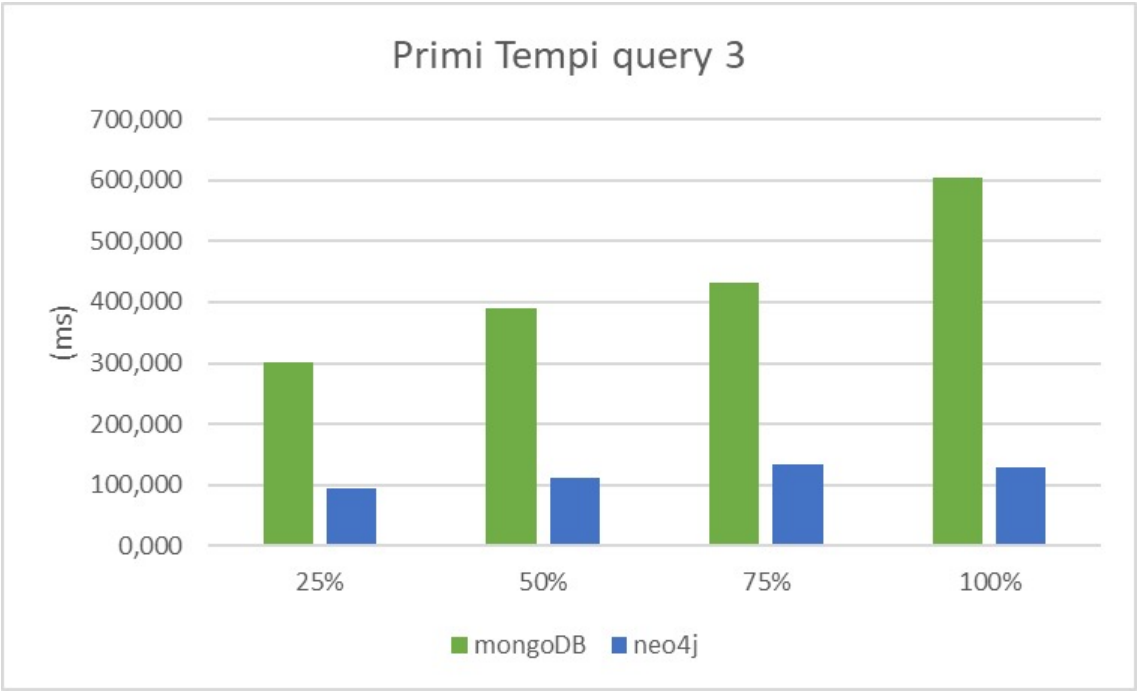
5.3.3 Query 3

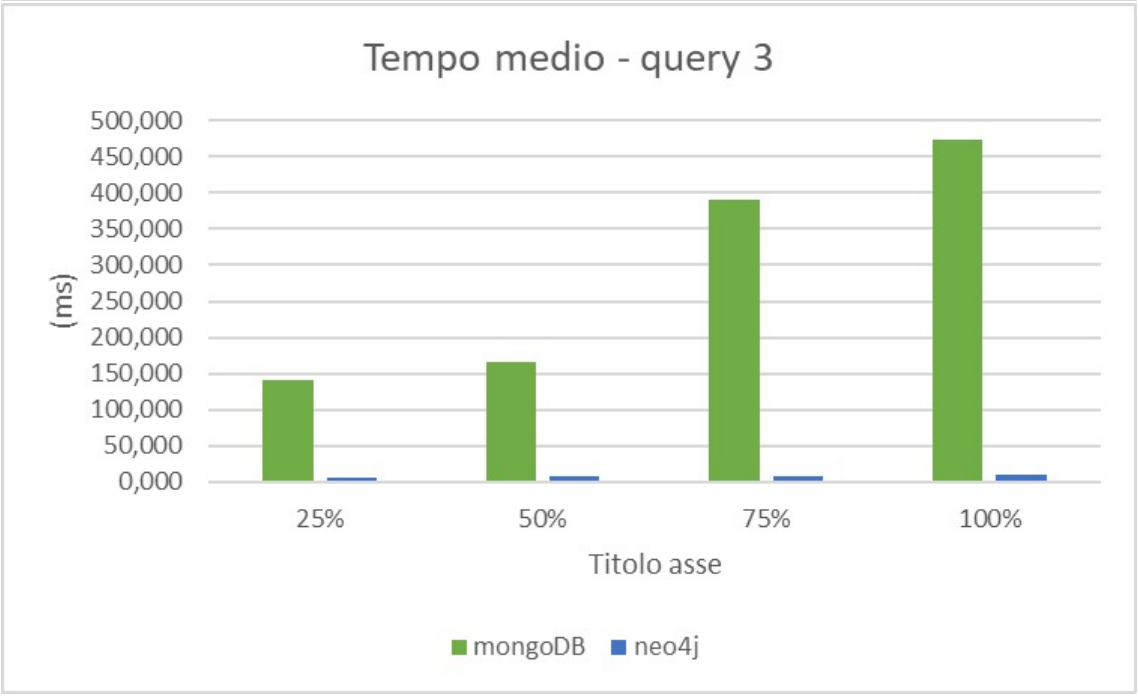
La terza query effettuata presenta una selezione condizionata con 2 WHERE su tre entità, unite tramite 2 JOIN; di seguito le corrispondenti versioni:

```
# SELECT *
# FROM calls
# JOIN people
# ON calls.CallingNbr = people.Number
# JOIN cells
# ON calls.CellSite = cells.CellSite
# WHERE StartDate >= 1580083200 AND EndDate < 1672473600
# MongoDB
query = [
{"$match": {"StartDate": {"$gte": start_search,
"$lt": end_search}}},
{"$lookup": {"from": "people",
"localField": "CallingNbr",
"foreignField": "Number",
"as": "Calling"}},
{"$lookup": {"from": "cells",
"localField": "CellSite",
"foreignField": "CellSite",
"as": "Cell"}}
]
# Neo4j
query = "MATCH (p1:person)-[r1:is_calling]->(c:call) \
-[r2:is_done]->(ce:cell) \
WHERE c.StartDate >= 1580083200 \
AND c.StartDate < 1672473600 \
RETURN p1, r1, c, r2, ce"
```

I tempi di esecuzione misurati (in millisecondi) sono i seguenti:

	primi tempi			media	
	mongoDB	neo4j		mongoDB	neo4j
25%	302,605	93,847	25%	140,424	6,485
50%	389,631	110,817	50%	166,752	7,068
75%	433,007	133,490	75%	390,520	7,298
100%	603,295	129,480	100%	473,886	9,184


Primi Tempi query 3
700,000
600,000
500,000
400,000
300,000
200,000
100,000
0,000
(ms)
25%
50%
75%
100%
mongoDB
neo4j


Tempo medio - query 3
500,000
450,000
400,000
350,000
300,000
250,000
200,000
150,000
100,000
50,000
0,000
(ms)
25%
50%
75%
100%
Titolo asse
mongoDB
neo4j

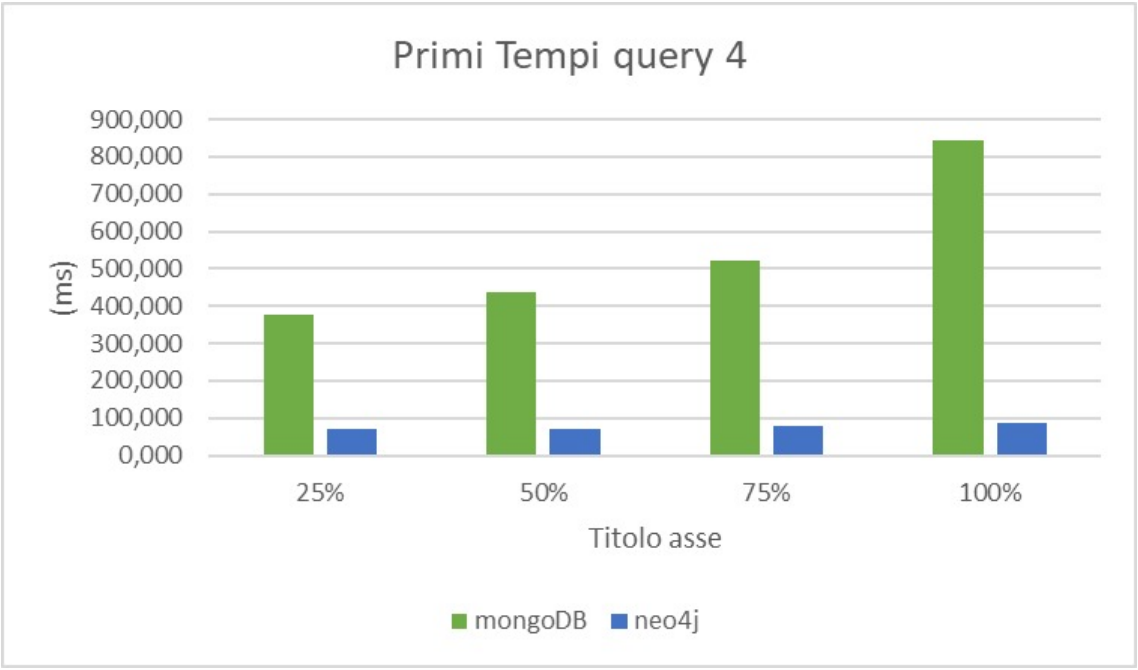
5.3.4 Query 4

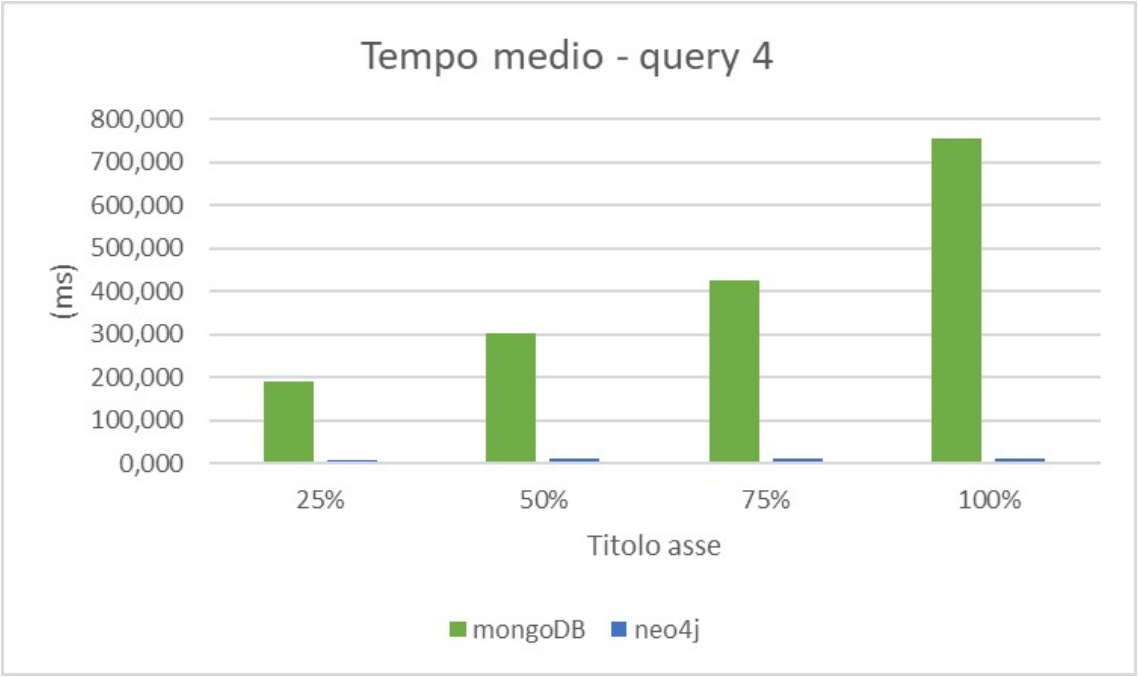
La quarta query effettuata presenta una selezione condizionata con 3 WHERE su tre entità, unite tramite 2 JOIN; di seguito le corrispondenti versioni:

```
# SELECT *
# FROM calls
# JOIN people
# ON calls.CallingNbr = people.Number
# JOIN cells
# ON calls.CellSite = cells.CellSite
# WHERE StartDate >= 1580083200 AND EndDate < 1672473600
# AND Duration > 900
# MongoDB
query = [
{"$match": {"StartDate": {"$gte": start_search,
"$lt": end_search},
"Duration": {"$gte": dur_search}}},
{"$lookup": {"from": "people",
"localField": "CallingNbr",
"foreignField": "Number",
"as": "Calling"}},
{"$lookup": {"from": "cells",
"localField": "CellSite",
"foreignField": "CellSite",
"as": "Cell"}}
]
# Neo4j
query = "MATCH (p1:person)-[r1:is_calling]->(c:call) \
-[r2:is_done]->(ce:cell) \
WHERE c.StartDate >= 1580083200 \
AND c.StartDate < 1672473600 \
AND c.Duration > 900 \
RETURN p1, r1, c, r2, ce"
```

I tempi di esecuzione misurati (in millisecondi) sono i seguenti:

	primi tempi			media	
	mongoDB	neo4j		mongoDB	neo4j
25%	375,323	68,847	25%	190,985	8,630
50%	435,677	72,817	50%	301,595	9,210
75%	523,557	79,490	75%	426,558	9,895
100%	845,298	85,480	100%	753,854	11,044


Primi Tempi query 4
900,000
800,000
700,000
600,000
500,000
400,000
300,000
200,000
100,000
0,000
(ms)
25%
50%
75%
100%
Titolo asse
mongoDB
neo4j


Tempo medio - query 4
800,000
700,000
600,000
500,000
400,000
300,000
200,000
100,000
0,000
(ms)
25%
50%
75%
100%
Titolo asse
mongoDB
neo4j

6 Analisi e Conclusione

Una prima evidenza è la sostanziale differenza tra i tempi di esecuzione sia in un ambiente con memoria cache libera e in un ambiente che sfrutta quest'ultima.

mongodb mantiene un'efficienza nell'uso della cache nettamente inferiore e non costante rispetto neo4j. Osservando i risultati della prima query in relazione alle successive è possibile estrapolare i punti di forza dei due DBMS: nella prima query infatti è presente un'operazione su singola entità, limitando l'affermazione alle selezioni, MongoDB risulta più efficiente, rimane sotto la soglia dei 15 ms con tutti e 4 i dataset utilizzati, questo risultato è raggiungibile da neo4j solo con l'utilizzo della cache(parlando sempre della prima query), in altre condizioni ovvero quando si parla di piu entità e quindi relazioni tra di esse neo4j non ne risente (n'è per complessità dell'interrogazione, n'è per quantità di dati) e si trova sempre in netto vantaggio rispetto a mongodb che invece fatica molto con le aggregazioni (JOIN in termini di SQL), . Per la precisione, Neo4j si mantiene in ogni situazione proposta sotto la soglia degli 120 ms, mentre per MongoDB al di fuori della sua situazione favorevole raggiunge un massimo di 850 ms circa, Entrambi i database rispettano tempi di risposta al più sotto l'1 s, con un dataset di più di un milione di istanze di entità, questo ci permette di dire che rispettano l'obiettivo dei DB NoSQL di gestire grandi quantità di dati in tempi più che accettabili, ovviamente con una gestione diversa e relativa al caso d'uso, infatti possiamo confermare che:

- Mongodb: è più adatto a quantità di dati non correlati o con poche relazioni e interrogazioni con bassa complessità per sfruttare al meglio il paradigma key-value dei db document-oriented. . Neo4j: è più adatto in situazione in cui è necessaria una maggiore responsività nella ricerca di dati strettamente correlati tra loro,perchè riesce a sfruttare appieno l'organizzazione del database a grafo con l'utilizzo delle relative operazioni tipiche di ricerca su tale struttura dati.